## Useful Packages

Some of these packages may NOT be included in your installation. Whenever you need to install a package, you need to use the Miniforge prompt or temrinal window, **NOT Python itself**. The Miniforge Prompt window can be reached through the Windows Start Menu folder for Miniforge3 or right clicking and opening a terminal from Applications > Utilities > Terminal on Mac.

Installing packages known to conda can be done with the `conda install <package name>` command in your Miniforge Prompt window. Otherwise you may need to use a different manager like `pip install <package name>`.

More information about managing packages in Python is available here.

**Data Packages**

- NumPy for numerical computation in Python
- scikit-learn for data analysis and machine learning
- Polars for dataframes designed for large-scale data processing performance
- DuckDB for creating a SQL database

**Other Utilities**

- Beautiful Soup for parsing HTML etc
- NLTK for text analysis
- Pillow for Images
- JobLib or Multiprocessing for running parallel/concurrent jobs

## Conda envs

conda provides the option to create separate Python environments inside your installation. All of the code below should be run in the Miniforge Prompt or Terminal:

(PC) Start Menu > Miniforge3 > Miniforge Prompt
(Mac) Finder > Applications > Utilities > Terminal

`conda create --name myenv python=3.5` creates an environment called `myenv` with Python version 3.5 instead of your main installation version.

`conda activate myenv` makes this environment active. From here you can install packages, and open software (e.g. `spyder` will open spyder after installation).

`conda deactivate` deactivates the active environment and returns to your base environment.

Conda environments are a great place to test out code, or run code that has very specific requirements. It's generally a good idea to be careful about vastly changing your environment (e.g. upgrading to a new version of Python), because it can break your project code! Environments provide a great way to test before making the change in your main environment.

Read more about conda environments

## Numpy

Numpy provides the mathematical functionality (e.g. large arrayes, linear algebra, random numbers, etc.) for many popular statistical and machine learning tasks in Python. This is a dependency for many of the packages we discuss below, including pandas. One of the foundational objects in numpy is the array:

In [1]:

```
import numpy as np
import pandas as pd
a_list = [[1,2],[3,4]] #list of ROWS
an_array = np.array(a_list, ndmin = 2)
a_dataframe = pd.DataFrame(a_list)
```

In [2]:

```
a_list
```

Out[2]:

```
[[1, 2], [3, 4]]
```

In [3]:

```
an_array
```

Out[3]:

```
array([[1, 2],
       [3, 4]])
```

In [4]:

```
a_dataframe
```

Out[4]:

| | **0** | **1** |
|---|---|---|
| **0** | 1 | 2 |
| **1** | 3 | 4 |

However, arrays in numpy are constrained to a single data type, unlike lists or DataFrames.

In [5]:

```
import numpy as np
import pandas as pd
a_list = [[1,"cat"],[3,"dog"]] #list of ROWS
an_array = np.array(a_list, ndmin = 2)
a_dataframe = pd.DataFrame(a_list)
```

In [6]:

```
pd.DataFrame(a_list).dtypes
```

Out[6]:

```
0     int64
1    object
dtype: object
```

In [7]:

```
pd.DataFrame(an_array).dtypes
```

```
Out[7]:
0    object
1    object
dtype: object
```

We can use numpy to do many numerical tasks, for example creating random values or matrices/DataFrames:

```
In [8]:
np.random.rand(2,2)
```

```
Out[8]:
array([[0.30901405, 0.91540236],
       [0.17105084, 0.09147348]])
```

```
In [9]:
np.zeros((3,4))
```

```
Out[9]:
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

```
In [10]:
np.ones((4,3,2))
```

```
Out[10]:
array([[[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]]])
```

Arrays make many mathematical operations easier than base Python. For example if we want to add a single value to every element of a list, we could try:

```
In [11]:
[1,2]+3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[11], line 1
----> 1 [1,2]+3

TypeError: can only concatenate list (not "int") to list
```

To accomplish this in base Python, we instead need to use a comprehension (maybe even with an if statement if the data types vary!):

```
In [12]:
base_list = [1,2]
```

```
[k+3 for k in base_list]
```

Out[12]:

```
[4, 5]
```

In [13]:

```
base_list = [1,2,"three"]
[k+3 for k in base_list if type(k)==int] #in this case we can use "int" because all values are inte
[k+3 for k in base_list if str(k).isnumeric()] #note that the .isnumeric() method is only available
```

Out[13]:

```
[4, 5]
```

With numpy arrays, we can use:

In [14]:

```
np.array([1,2])+3
```

Out[14]:

```
array([4, 5])
```

or

In [15]:

```
arr = np.array([1,2])
arr += 3
print(arr)
```

```
[4 5]
```

Since the pandas dataframes are built on numpy arrays:

In [16]:

```
pd.DataFrame([1,2])+3
```

Out[16]:

| | **0** |
|---|---|
| **0** | 4 |
| **1** | 5 |

`SciPy` adds an array of mathematical and statistical functions that work with `numpy` objects.

## Pandas and Data Visualization Packages

See our dedicated lesson on Pandas (linked above).

## scikit-learn

scikit-learn provides a consolidated interface for machine learning in Python:

- functions for splitting data into training and testing components
- cross validation for model tuning
- supervised and unsupervised modeling
- model fit assessment and comparison

Read more about using sklearn. Digging into the application of machine learning is beyond the scope of our workshop series.

The following example comes from Scitkit-learn's Linear Regression Example page

In [ ]:

```
# !conda install scikit-learn

from sklearn import linear_model, datasets
import matplotlib.pyplot as plt

# Load the diabetes dataset
diabetes_X, diabetes_y = datasets.load_diabetes(return_X_y=True)

# Use only one feature
diabetes_X = diabetes_X[:, np.newaxis, 2]

# Split the data into training/testing sets
diabetes_X_train = diabetes_X[:-20]
diabetes_X_test = diabetes_X[-20:]

# Split the targets into training/testing sets
diabetes_y_train = diabetes_y[:-20]
diabetes_y_test = diabetes_y[-20:]

# Create linear regression object
regr = linear_model.LinearRegression()

# Train the model using the training sets
regr.fit(diabetes_X_train, diabetes_y_train)

# Make predictions using the testing set
diabetes_y_pred = regr.predict(diabetes_X_test)

# The coefficients
print("Coefficients: \n", regr.coef_)
```

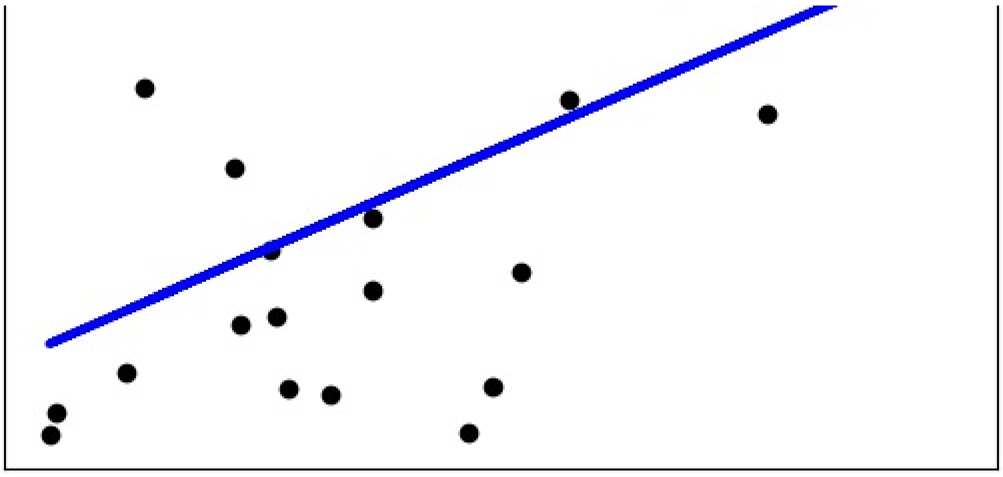
```
Coefficients:
 [938.23786125]
```

In [ ]:

```
# Plot outputs
plt.scatter(diabetes_X_test, diabetes_y_test, color="black")
plt.plot(diabetes_X_test, diabetes_y_pred, color="blue", linewidth=3)

plt.xticks(())
plt.yticks(())

plt.show()
```



## Polars (dataframes for large-scale data processing)

The Polars library offers an alternative to Pandas dataframes that often performs much faster and uses less RAM for dataframe operations. Polars is built in Rust while Pandas is built on NumPy, which has lower performance and higher memory use than Rust. Polars is also able to efficiently run parallel processes, adding to its improved performance.

In [ ]:

```
# !pip install polars

import sys
import time
import polars as pl
import pandas as pd

# how long does it take Polars to load in a CSV?
start_pl = time.time()
df_pl = pl.read_csv("protest_data.csv")
end_pl = time.time()
print(f'Seconds for Polars to load in the protest data CSV: {end_pl-start_pl}')

# how long does it take Pandas to load in a CSV?
start_pd = time.time()
df_pd= pd.read_csv("protest_data.csv")
end_pd = time.time()
print(f'Seconds for Pandas to load in the protest data CSV: {end_pd-start_pd}')

# how much faster is Polars?
print(f'Polars is {round((end_pd-start_pd)/(end_pl-start_pl), 2)}x faster')
```

```
Seconds for Polars to load in the protest data CSV: 0.02615809440612793
Seconds for Pandas to load in the protest data CSV: 0.1337299346923828
Polars is 5.11x faster
```

In [22]:

```
# compare the memory size of the Polars and Pandas dataframes
print(f'The Polars dataframe takes up {sys.getsizeof(df_pl)} bytes.')
print(f'The Pandas dataframe takes up {sys.getsizeof(df_pd)} bytes.')
```

```
The Polars dataframe takes up 48 bytes.
The Pandas dataframe takes up 28690152 bytes.
```

## DuckDB (for creating a SQL Database)

DuckDB is a great library for setting up a SQL-database with Python. It does not have any dependencies and is very memory efficient, making it a faster alternative to PostgreSQL, MySQL, or SQLite.

In [25]:

```
#!pip install duckdb
import duckdb

duckdb.read_csv("protest_data.csv")
duckdb.sql("SELECT id, country, ccode, year, protest FROM 'protest_data.csv' WHERE YEAR > 2010 LIMI
```

Out[25]:

```
┌───────────┬─────────┬───────┬───────┬─────────┐
│    id     │ country │ ccode │ year  │ protest │
│   int64   │ varchar │ int64 │ int64 │  int64  │
├───────────┼─────────┼───────┼───────┼─────────┤
│ 202011001 │ Canada  │    20 │  2011 │       1 │
│ 202012001 │ Canada  │    20 │  2012 │       1 │
│ 202013000 │ Canada  │    20 │  2013 │       0 │
│ 202014000 │ Canada  │    20 │  2014 │       0 │
│ 202015001 │ Canada  │    20 │  2015 │       1 │
│ 202016001 │ Canada  │    20 │  2016 │       1 │
│ 202016002 │ Canada  │    20 │  2016 │       1 │
│ 202016003 │ Canada  │    20 │  2016 │       1 │
│ 202016004 │ Canada  │    20 │  2016 │       1 │
│ 202016005 │ Canada  │    20 │  2016 │       1 │
├───────────┴─────────┴───────┴───────┴─────────┤
│ 10 rows                             5 columns │
└───────────────────────────────────────────────┘
```

## BeautifulSoup (for parsing HTML or XML data)

Python's built-in `urllib.request` package makes it relatively easy to download the underlying html from a web page. Note that the `from <package> import <function>` notation used here allows you to selectively import only parts of a package as needed.

Be sure to check the terms of services for any website before scraping! We're scraping our own materials here to be safe!

In [ ]:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
page = urlopen("https://unc-libraries-data.github.io/Python/Intro/Introduction.html")  #The Python
html = page.read()
print(html[:300]) #print only the first 300 characters
```

```
b'<!DOCTYPE html>\n<html>\n<head><meta charset="utf-8" />\n\n<title>Introduction</title>\n\n<script
src="https://cdnjs.cloudflare.com/ajax/libs/require.js/2.1.10/require.min.js"></script>\n<script sr
c="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.0.3/jquery.min.js"></script>\n\n\n\n<style type
="text/css">\n    '
```

In [ ]:

```
soup=BeautifulSoup(html,"html.parser")
[x.text for x in soup.find_all("h2")] # find all h2 (second-level headers)
```

```
Out[ ]:
['Why Python?¶',
 'Getting Started¶',
 'Data Types and Variables¶',
 'Flow Control¶',
 'More Data Types¶',
 'Review¶',
 'Pseudocode and Comments¶',
 'User-defined Functions¶',
 'Coming up¶',
 'References and Resources¶']
```

## APIs

APIs (Application Programming Interfaces) provide a structured way to request data over the internet. APIs are generally a better option than web scraping because:

- they provided structured data instead of often inconsistent HTML or other formats you'll have to dig through to find relevant information
  - this often has the added benefit of not taxing a website as much as web scraping!
- they provide a clearer path to retrieving information with permission

An API call is just a specific type of web address that you can use Python to help you generate (or cycle through many options):
For example: https://api.weather.gov/points/35.9132,-79.0558
This link pulls up the National Weather Service information for a particular lat-long pair (for Chapel Hill). The forecast field leads us to a new link:
https://api.weather.gov/gridpoints/LWX/96,70/forecast

We can use Python to request and parse the content from these links, but often we can find a wrapper someone else has created to do some of that work for us!

Remember that we can install packages in the Miniforge Prompt or Terminal:

- (PC) Start Menu > Miniforge3 > Miniforge Prompt
- (Mac) Finder > Applications > Utilities > Terminal

Then run the following to install the package:
`pip install noaa_sdk`

```
In [30]:
# !pip install noaa_sdk

from noaa_sdk import NOAA
from time import sleep

n = NOAA()
forecast = n.points_forecast(35.9132,-79.0558, type='forecastGridData')
sleep(5) #pause for 5 seconds to prevent repeatedly using the API
```

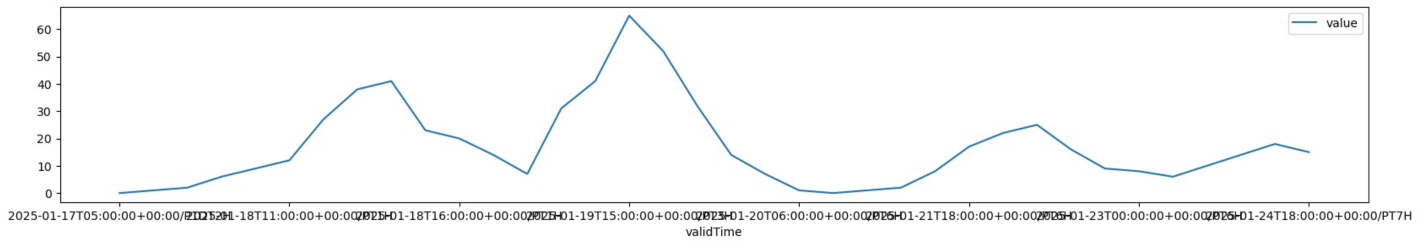
This provides us with a pretty complicated set of nested dictionaries that we can parse to find specific values:

```
In [ ]:
```

```
rain_chance = forecast["properties"]["probabilityOfPrecipitation"]["values"]
pd.DataFrame(rain_chance).plot.line(x="validTime",y="value", figsize=(20,3))
```

Out[ ]:

```
<Axes: xlabel='validTime'>
```



## NLTK (text analysis)

The Natural Language Toolkit ( `nltk` ) provides a wide array of tools for processing and analyzing text. This includes operations like splitting text into sentences or words ("tokenization"), tagging them with their part of speech, classification, and more.

Let's take the example sentence: "The quick brown fox jumps over the lazy dog." and convert it into individual words.

In [ ]:

```
import nltk
#the code below is necessary for word_tokenize and parts of speech to work
# nltk.download("punkt_tab")
# nltk.download('averaged_perceptron_tagger_eng')
sentence = "The quick brown fox jumps over the lazy dog."
words = nltk.word_tokenize(sentence)
print(words)
```

```
Requirement already satisfied: nltk in /Users/rolando/miniforge3/lib/python3.12/site-packages (3.9.
1)
Requirement already satisfied: click in /Users/rolando/miniforge3/lib/python3.12/site-packages (fro
m nltk) (8.1.8)
Requirement already satisfied: joblib in /Users/rolando/miniforge3/lib/python3.12/site-packages (fr
om nltk) (1.4.2)
Requirement already satisfied: regex>=2021.8.3 in /Users/rolando/miniforge3/lib/python3.12/site-pac
kages (from nltk) (2024.9.11)
Requirement already satisfied: tqdm in /Users/rolando/miniforge3/lib/python3.12/site-packages (from
nltk) (4.66.5)
[nltk_data] Downloading package punkt_tab to
[nltk_data]     /Users/rolando/nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger_eng to
[nltk_data]     /Users/rolando/nltk_data...
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog', '.']
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger_eng.zip.
```

Now we can extract the part of speech for each word in the sentence. Note that this function, like many of the functions in NLTK, uses machine learning to classify each word and therefore may have some level of error!

In [38]:

```
nltk.pos_tag(words)
```

Out[38]:

```
[('The', 'DT'),
 ('quick', 'JJ'),
 ('brown', 'NN'),
 ('fox', 'NN'),
 ('jumps', 'VBZ'),
 ('over', 'IN'),
 ('the', 'DT'),
 ('lazy', 'JJ'),
 ('dog', 'NN'),
 ('.', '.')]
```

The meaning of these parts of speech tags are available below:

In [28]:

```
# nltk.download('tagsets')
# nltk.help.upenn_tagset()
```

**Read more about getting data for Text and Data Mining projects via the Libraries.**

## PIL (Pillow)

Pillow is the updated version of the old Python Imaging Library (PIL), which provides fundamental tools for working with images. Pillow can work with a many common formats (some of which may require extra packages or other dependencies) to automate a wide variety of image transformations.

Note: While `pillow` is how you install the package, you import functions with `import PIL` .

Both `display` and `imshow` can be used to preview the image. Currently, `display` returns some errors due to the image's modes, so we're using `imshow` to avoid these errors. Read more about modes.

In [ ]:

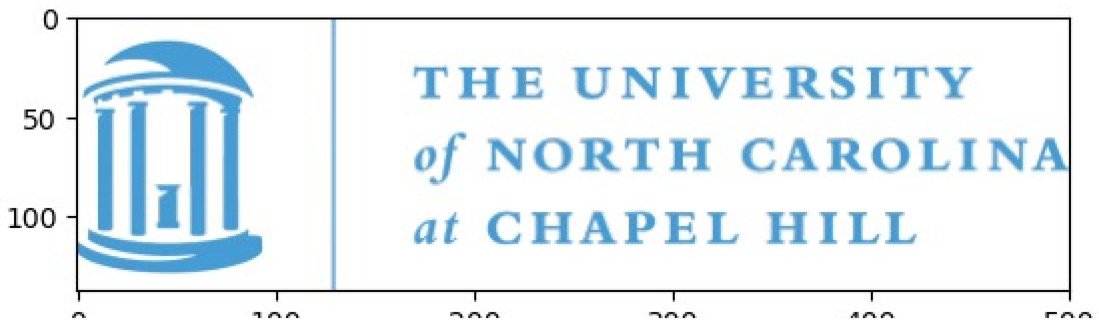
```
from PIL import Image
from urllib.request import urlretrieve
from matplotlib.pyplot import imshow
from IPython.display import display


#downloading an image locally
urlretrieve("https://identity.unc.edu/wp-content/uploads/sites/885/2019/01/UNC_logo_webblue-e151794
           "UNC_logo.png")

UNC = Image.open("UNC_logo.png")
#note: // divides two numbers and rounds the result down to get an integer
UNC_gray = UNC.convert('LA').resize((UNC.width//2,UNC.height//2))
imshow(UNC)
```

Out[ ]:

```
<matplotlib.image.AxesImage at 0x28801cda0>
```

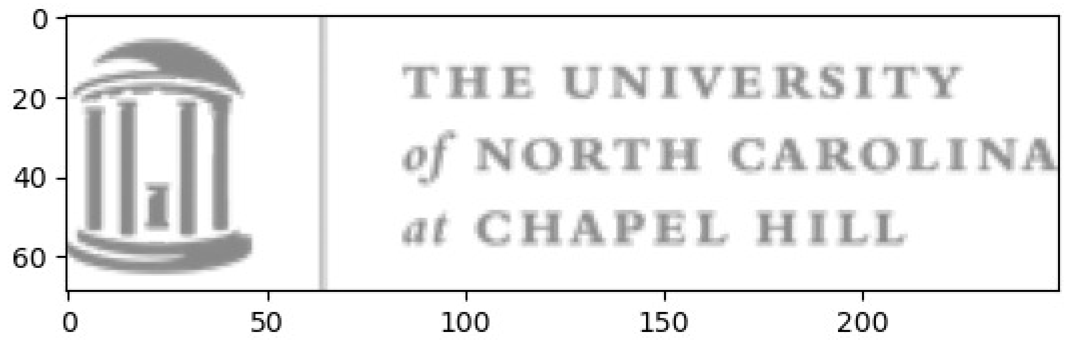


Mode "LA" is grayscale, preserving transparent image areas.

```
In [40]:
```

```python
imshow(UNC_gray)
```

```
Out[40]:
<matplotlib.image.AxesImage at 0x288ac6720>
```



## Parallel Processing with joblib

As you move into more complicated processes in Python (or applying code to a wide variety of objects or files), processing time can become a major factor. Fortunately, most modern laptops have multiple processor cores that can do separate things at the same time. Python only uses **one core** by default. If you have a set of loops that don't depend on each other (e.g. processing lots of files one after another), you could split up your many loop iterations between processors to greatly increase speed.

The `joblib` package provides a straightforward way to split loops up between your computers cores for faster performance on complicated code. Note that parallelization may not benefit you much or may even hurt you for very quick jobs because setting up and consolidating information from separate cores creates overhead costs.